# Apollo's Scalability

Ivan Mamontov

May 27, 2015

## Contents

## Revision History

| Revision | Date | Author(s) | Description |
|---|---|---|---|
| 1.0 | 27.04.15 | IM | Initial draft |

**Abstract**

This paper characterizes the performance of Apollo on a quite modern Intel®Core$^{TM}$ i7 CPU with Hyper-Threading Technology(HT). Using the performance counters provided by Intel, it was revealed that: (1) HT can't improve throughput of Apollo; (2) The resource contentions within CPU are the major reason of pipeline inefficiency, which prevents better performance promised by HT; (3) CPU utilization isn't a good estimate of the true load on the system with enabled HT.

## 1 Theory

Current trends in microprocessor design have made high resource utilization a key requirement for achieving good performance. For example, while deeper pipelines have led to 3 GHz processors, each new generation of microarchitecture technology comes with increased memory latency and a decrease in relative memory speed. This results in the processor spending a significant amount of time waiting for the memory system to fetch data. This "memory wall" problem continues to remain a major bottleneck[1].

Over the years, a number of multithreading techniques have been employed to hide this memory latency. One approach is simultaneous multithreading (SMT), which exposes more parallelism to the processor by fetching and retiring instructions from multiple instruction streams, thereby increasing processor utilization. This allows the CPU to hold the state of different threads and then switch back and forth on a nanosecond time scale. For example, if one of the processes needs to read a word from memory (which takes many clock cycles), a CPU can just switch to another thread. SMT does not offer true parallelism, i.e. only one process at a time is running, but thread-switching time is reduced to the order of a nanosecond[2]. Price and performance benefits make SMT a common design choice as, for example, in Intel's Nehalem microarchitecture, where it is called Hyper-Threading (HT) where the physical core contains a mixture of resources, some of which are shared between threads[3]:

- replicated resources for each thread, such as register state, return stack buffer (RSB), and the instruction queue;
- partitioned resources tagged by the thread number, such as load buffer, store buffer, and reorder buffer;

- shared resources such as:
  - Instruction fetchers, decoders etc are cooperatively shared between HTs. That is the fetch/decode path alternates between HT threads on each CPU tick. This halves the rate at which instructions can be input at the top of the pipe.
  - L1/L2 and TLB space, and for branch prediction resources.
  - Reservation Station (RS), where the uops wait until their inputs are available, space in the Reorder Buffer, where the uops wait until they can be retired sufficient load and store buffers in the case of memory related uops
- shared resources unaware of the presence of threads, such as execution units.

Because instructions from several threads execute simultaneously, threads compete every cycle for all common hardware resources, such as functional units, instruction queues, renaming registers, caches, and TLBs. Since threads may differ widely in their hardware requirements, some threads may interact poorly when co-scheduled onto the processor. For example, two threads with large cache footprints may cause inter-thread cache misses, leading to low instruction throughput for the machine as a whole. Conversely, threads with complementary resource requirements may coexist on the processor without excessive interference, thereby increasing utilization; for example, integer-intensive and FP-intensive threads should execute well together, since they utilize different functional units. Consequently, thread scheduling decisions have the potential to affect performance, either improving it by co-scheduling threads with complementary hardware requirements, or degrading it by co-scheduling threads with identical hardware needs[4].

## 2 Practice

### 2.1 Problem definition

The key principle in achieving acceptable server performance is to avoid running the server at maximum hardware resource utilization on a regular basis, so it is important to establish acceptable thresholds for hardware resource utilization to provide a reserve capacity for peak utilization periods. In this case HT can be quite a misleading, for example, if CPU with HT shows 40% CPU usage, this does not mean that this CPU can handle more than 2 times more work. So when it comes to capacity planning, it's easy to be duped into thinking that CPU utilization is a good estimate of the true load on the system with enabled HT.

## 2.2 Experimental platform

To get a better understanding of how HT impacts performance, consider a single-socket Intel Core i7 processor-based system with four cores (Intel Xeon will have the same result as it uses the same microarchitecture).

- OS: RHEL 7 (kernel 3.10) with complete fair queuing (CFQ) scheduler.
- CPU: 2.80GHz Intel®Core$^{TM}$ i7 CPU 860 processor with 4 physical cores and 8 hyper-threads. This is a Nehalem - quite old hardware but similar to that used in production.
- Memory: 16GB of dual channel DDR3 memory at 1600MHz.
- JVM: Oracle JDK 1.7.0_75 with production-like settings.
- Apollo: last stable release with production-like index.

## 2.3 Benchmark

Let's assume a jagger[6] test which is capable to produce a linearly increasing keyword search load with facets and user filters as in production. There are two cases to check:

- HT is disabled, Apollo uses four hardware threads;
- HT is enabled, Apollo uses eight software threads;

Figure 1 shows the result of these experiments. In both cases we have the same upper bound on system throughput. Since there is no significant difference between the results most likely this is throughput of the shared resource when its utilization reaches one.

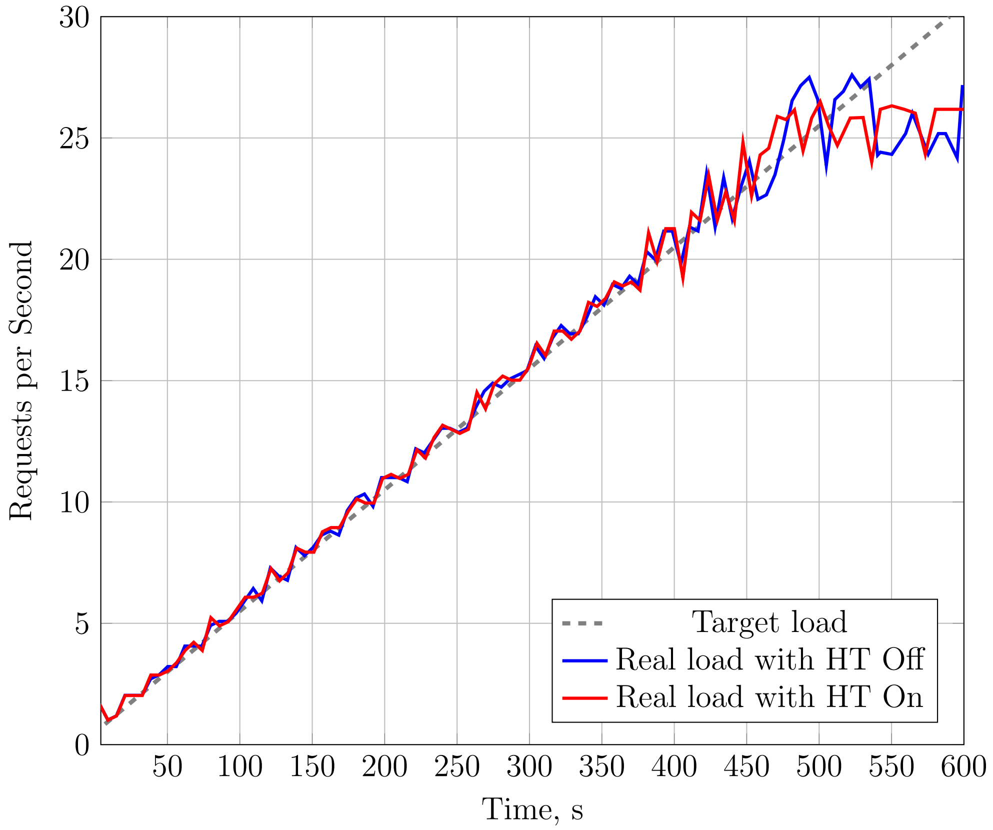

Figure 1: Apollo's throughput with and without HT

## 3 Analysis

### 3.1 CPU utilization

CPU utilization grows linearly until the processor reaches about 70 percent utilization without HT and 40 percent with HT as illustrated in Figure 2. At that point, CPU utilization becomes exponential and rises quickly. The results are more than illustrative – HT does not actually give twice more power in our case. As we can see at 50% of CPU utilization in HT mode, the CPU becomes overloaded because the two threads running in the same core are competing for the same resources. If each of the two threads gets only half the amount of a limiting resource then it will run at half speed, and the advantage of HT is completely gone. Two threads running at half speed is certainly not better than a single thread running at full speed.

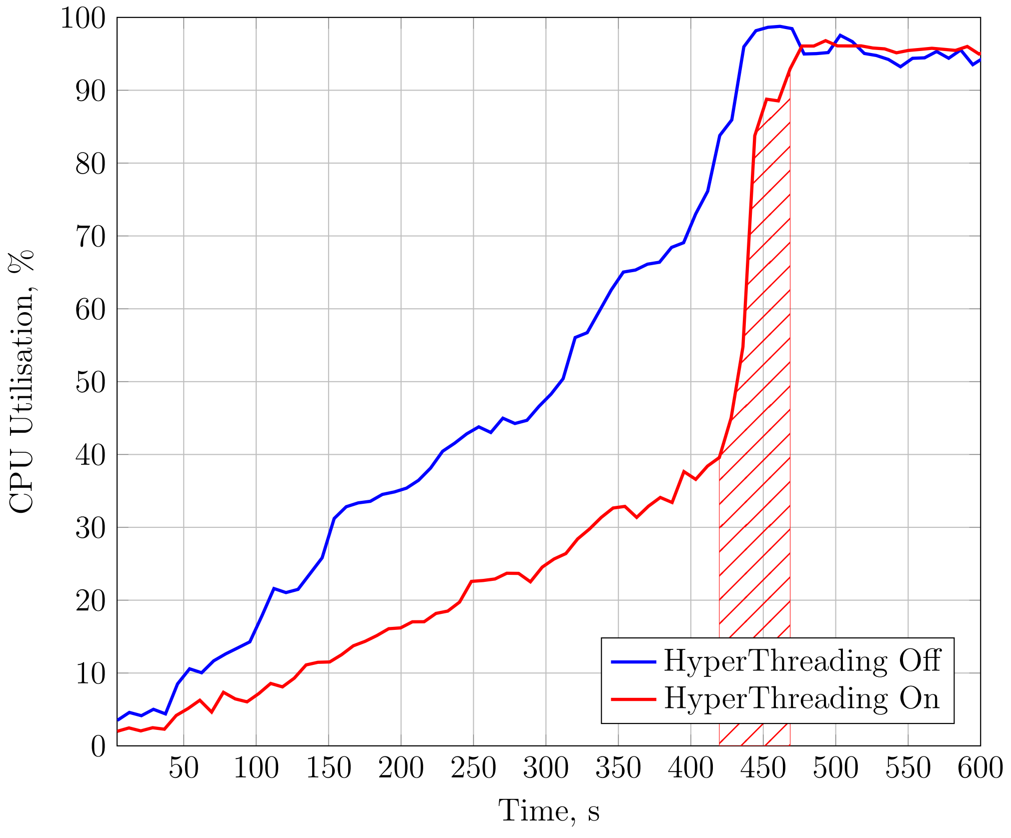

Figure 2: Dependency of the average CPU utilization on linearly increasing keyword search load

HT has implications for the operating system because each thread appears to the operating system as a separate CPU. So if there is only enough work to keep each hardware thread busy at a certain point in time the operating system will have to place active threads on cores before scheduling on threads on the same core. Unfortunately, sometimes it happens that the operating system lets two threads run in the same processor core, with the other CPU completely idle. This choice is far less efficient than using one thread on each CPU. Figure 3 shows this behavior during the test when utilization starts to increase rapidly(red-shaded area on figure 2). Each horizontal line on this timechart represents logical CPU(from 0 to 7). Each bar on this chart represents a scheduler timeslice. It is obvious that while the first core woks with two threads, ”the last core” is idle. It is the responsibility of the operating system to avoid such situations, but unfortunately, in order to avoid unnecessary CPU migrations scheduler decides to keep thread on overloaded core.

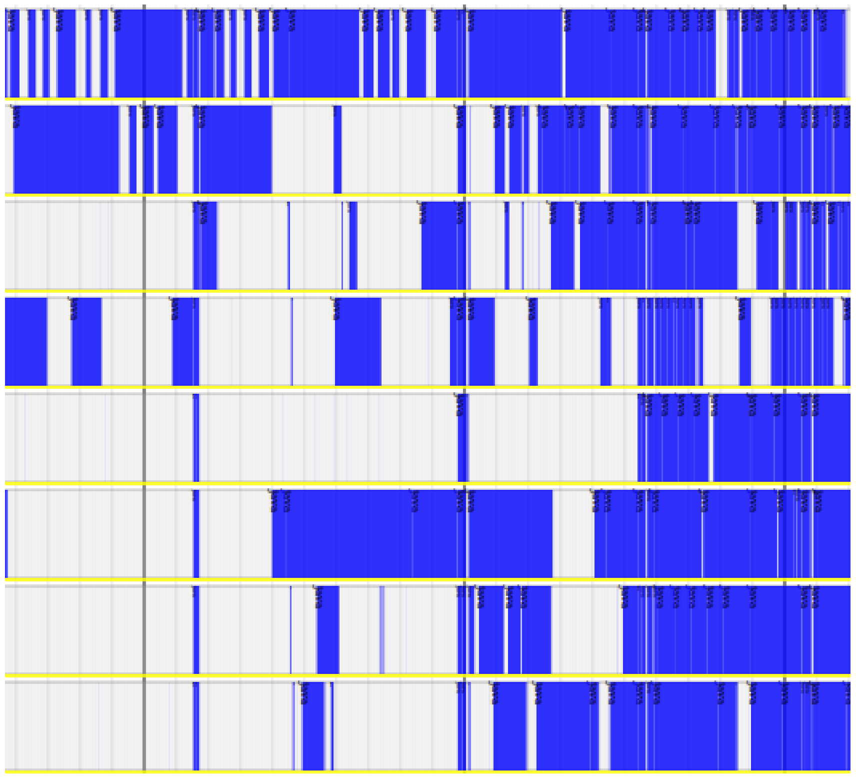

Figure 3: CPU scheduler behavior during the test in the red zone

## 3.2 Analyzing bottlenecks

In order to better understand the bottleneck it will be useful to use Performance Monitoring Unit (PMU) which is a set of special-purpose registers built into modern microprocessors to store the counts of hardware-related activities such as instruction cycles, cache hits, cache misses, branch misses and many others. Compared to software profilers, hardware counters provide low-overhead access to a wealth of detailed performance information related to CPU's functional units, caches and main memory etc. Another benefit of using them is that no source code modifications are needed in general.

The first thing which is worth paying attention to is shared resources. If the performance is limited by any of the shared resources, for example highest-level cache, which is called before accessing memory, is usually referred to as the last level cache(LLC), then the total performance is not increased by HT. Actually, in the worst cases the total performance is decreased by HT because this resources are wasted when the two threads compete for the same resources. Figure 4 presents the last level cache misses per 1000 instructions.

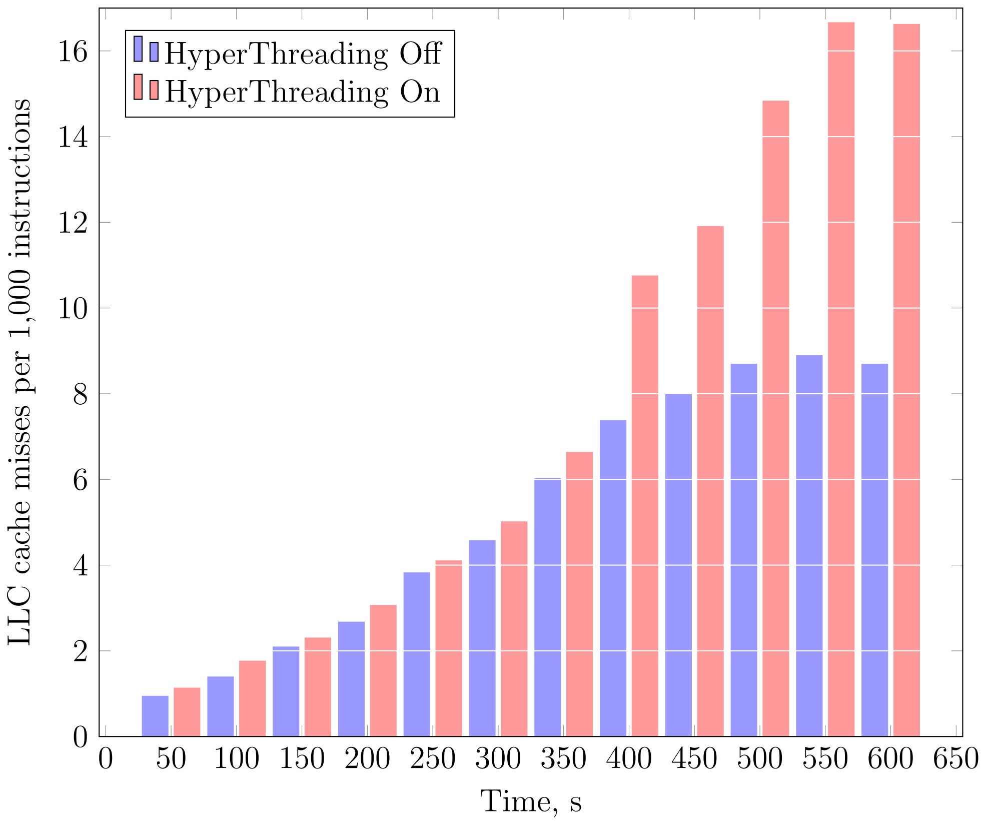

Figure 4: LLC cache misses per 1,000 instructions

The miss rate increases significantly in case of enabled HT which is caused by the extremely large memory footprint, which does not fit in the LLC cache.

Another shared resources are instruction fetchers and decoders. There are a lot of stalled frontend instructions in frontend - Figure 5.

```
     74360.715495   task-clock (msec)         #    3.835 CPUs utilized
   84,400,226,408   stalled-cycles-frontend   #   55.59% frontend cycles idle
   41,616,239,021   stalled-cycles-backend    #   27.41% backend  cycles idle
  170,735,558,048   instructions              #    1.12  insns per cycle
                                              #    0.49  stalled cycles per insn
      716,316,970   branch-misses             #    2.33% of all branches
```

Figure 5: IPC and frontend stalls without HT

The saddest thing is that it gets worse with HT, as shown in Figure 6. Following up on this difference will be interesting, but outside the scope of this work. This observation is another example why HT can't improve throughput of Apollo.

```
    137014.930439   task-clock (msec)         #    7.839 CPUs utilized
  170,689,428,657   stalled-cycles-frontend   #   70.69% frontend cycles idle
   47,010,900,540   stalled-cycles-backend    #   19.47% backend  cycles idle
  187,613,979,340   instructions              #    0.78  insns per cycle
                                              #    0.91  stalled cycles per ins
      915,996,259   branch-misses             #    2.70% of all branches
```

Figure 6: IPC and frontend stalls with HT

Usually, stalled cycles are cycles where the processor is waiting for something (memory to be feed after executing a load operation for example) and doesn't have any other stuff to do. Moreover, the frontend part of the CPU is the piece of hardware responsible to fetch and decode instructions (convert them to uops) where as the backend part is responsible to effectively execute the uops.

## 4 Conclusion

Obviously, it can be quite difficult to predict whether HT is good or bad for a particular application. The only safe way of answering this question is to test it. Ideally, we should test our application on several different microprocessors with several different data sets with HT turned on and off. Throughput may be improved only if you can multithread your workload and that the workload was previously resource limited by something other than CPU execution resources. If CPU resources are the limiting factor, having two independent instruction streams competing for them won't help with either throughput or latency.

## References

[1] Subhash Saini, Haoqiang Jin The Impact of Hyper-Threading on Processor Resource Utilization in Production Applications

[2] Andrew S. Tanenbaum Modern Operating Systems. Fourth Edition

[3] David Levinthal Performance Analysis Guide for Intel® Core™ i7 Processor and Intel® Xeon™ 5500 processors `https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf`

[4] Sujay Parekh, Susan Eggers Thread-Sensitive Scheduling for SMT Processors

[5] Agner Fog How good is hyperthreading? Available at `http://agner.org/optimize/blog/read.php?i=6`

[6] `https://jagger.griddynamics.net/`

[7] Garrett Drysdale, Antonio C. Valles, Matt Gillespie Performance Insights to Intel® Hyper-Threading Technology

[8] Wei Huang, Jiang Lin, Zhao Zhang, J. Morris ChangPerformance Characterization of Java Applications on SMT Processors

[9] Intel Corp. VTune performance analyzer. `http://www.intel.com/software/products/vtune/`

[10] Intel Corp. Intel® 64 and IA-32 Architectures Optimization Reference Manual `http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html`

[11] Brendan D. Gregg Active Benchmarking http://www.brendangregg.com/activebenchmarking.html

[12] `https://perf.wiki.kernel.org/index.php/Main_Page`

[13] Edward D. Lazowska, John Zahorjan Quantitative System Performance `http://homes.cs.washington.edu/~lazowska/qsp/Images/Chap_05.pdf`